


GALAHAD BSC

USER DOCUMENTATION GALAHAD Optimization Library version 5.1

1 SUMMARY

Given matrices **A** and diagonal **D**, this package forms the **Schur complement S** $= \mathbf{ADA}^T$ in sparse co-ordinate format. Full advantage is taken of any zero coefficients in the matrices **A**.

ATTRIBUTES — Versions: `GALAHAD_BSC_single, GALAHAD_BSC_double`. **Uses:** `GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_SMT, GALAHAD_QPT, GALAHAD_SPECFILE`, **Date:** October 2013. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

The package is available with single, double and (if available) quadruple precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the `USE` statement

```
USE GALAHAD_BSC_single
```

with the obvious substitution `GALAHAD_BSC_double`, `GALAHAD_BSC_quadruple`, `GALAHAD_BSC_single_64`, `GALAHAD_BSC_double_64` and `GALAHAD_BSC_quadruple_64` for the other variants.

If it is required to use more than one of the modules at the same time, the derived types `SMT_type`, `QPT_problem_type`, `BSC_time_type`, `BSC_control_type`, `BSC_inform_type` and `BSC_data_type` (§2.3) and the subroutines `BSC_initialize`, `BSC_form`, `BSC_terminate`, (§2.4) and `BSC_read_specfile` (§2.6) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

The input matrix **A** may be stored in a variety of input formats.

2.1.1 Dense storage format

The matrix **A** is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Component $n*(i-1)+j$ of the storage array `A%val` will hold the value a_{ij} for $i = 1,\ldots,m$, $j = 1,\ldots,n$.

2.1.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry of **A**, its row index i, column index j and value a_{ij} are stored in the l-th components of the integer arrays `A%row`, `A%col` and real array `A%val`, respectively. The order is unimportant, but the total number of entries `A%ne` is also required.

2.1.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row $i+1$. For the i-th row of **A**, the i-th component of a integer array `A%ptr` holds the position of the first entry in this row, while `A%ptr` $(m+1)$ holds the total number of entries plus one. The column indices j and values a_{ij} of the entries in the i-th row are stored in components $l =$ `A%ptr`(i), ...,`A%ptr` $(i+1)-1$ of the integer array `A%col`, and real array `A%val`, respectively.

For sparse matrices, this scheme almost always requires less storage than its predecessor.

2.2 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords REAL(rp_) and INTEGER(ip_), where rp_ and ip_ are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default REAL for the single precision versions, DOUBLE PRECISION for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to rp_ = real32, rp_ = real64 and rp_ = real128 respectively as defined by the fortran iso_fortran_env module. The latter are default (32-bit) and long (64-bit) integers, and correspond to ip_ = int32 and ip_ = int64, respectively, again from the iso_fortran_env module.

2.3 The derived data types

Four derived data types are accessible from the package.

2.3.1 The derived data type for holding matrices

The derived data type SMT_TYPE is used to hold the matrix **A**. The components of SMT_TYPE used here are:

type is a rank-one allocatable array of type default CHARACTER, that is used to indicate the storage scheme used. If the dense storage scheme (see §2.1.1), is used, the first five components of type must contain the string DENSE. For the sparse co-ordinate scheme (see §2.1.2), the first ten components of type must contain the string COORDINATE, and for the sparse row-wise storage scheme (see §2.1.3), the first fourteen components of type must contain the string SPARSE_BY_ROWS.

For convenience, the procedure SMT_put may be used to allocate sufficient space and insert the required keyword into type. For example, if A is of derived type SMT_type and we wish to use the co-ordinate storage scheme, we may simply

```
CALL SMT_put( A%type, 'COORDINATE', istat )
```

See the documentation for the GALAHAD package SMT for further details on the use of SMT_put.

ne is a scalar variable of type INTEGER(ip_), that holds the number of matrix entries.

val is a rank-one allocatable array of type REAL(rp_) and dimension at least ne, that holds the values of the entries. Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.

row is a rank-one allocatable array of type INTEGER(ip_), and dimension at least ne, that may hold the row indices of the entries. (see §2.1.2).

col is a rank-one allocatable array of type INTEGER(ip_), and dimension at least ne, that may hold the column indices of the entries (see §2.1.2–2.1.3).

ptr is a rank-one allocatable array of type INTEGER(ip_), and dimension at least m + 1, that may hold the pointers to the first entry in each row (see §2.1.3).

2.3.2 The derived data type for holding control parameters

The derived data type BSC_control_type is used to hold controlling data. Default values may be obtained by calling BSC_initialize (see §2.4.1), while components may also be changed by calling GALAHAD_BSC_read_spec (see §2.6.1). The components of BSC_control_type are:

error is a scalar variable of type INTEGER(ip_), that holds the stream number for error messages. Printing of error messages in BSC_solve and BSC_terminate is suppressed if error ≤ 0. The default is error = 6.

out is a scalar variable of type INTEGER(ip_), that holds the stream number for informational messages. Printing of informational messages in BSC_solve is suppressed if out $<$ 0. The default is out = 6.

print_level is a scalar variable of type INTEGER(ip_), that is used to control the amount of informational output which is required. No informational output will occur if print_level $\leq$ 0. If print_level $=$ 1, a single line of output will be produced for each iteration of the process. If print_level $\geq$ 2, this output will be increased to provide significant detail of each iteration. The default is print_level = 0.

new_a is a scalar variable of type INTEGER(ip_), that is used to indicate how **A** has changed (if at all) since the previous call to BSC_form. Possible values are:

0 **A** is unchanged

1 the values in **A** have changed, but its nonzero structure is as before.

2 both the values and structure of **A** have changed.

3 the structure of **A** has changed, but only the structure of **S** (and not its values) is required.

The default is new_a = 2.

max_col is a scalar variable of type INTEGER(ip_), that specifies the maximum number of nonzeros in a column of **A** which is permitted when building the Schur-complement. Any negative value will be interpreted as m. The default is max_col = -1.

extra_space_s is a scalar variable of type INTEGER(ip_), that specifies how much extra space (if any) is to be allocated for the arrays that will hold **S** above that needed to hold the Schur complement. This may be useful, for example, if another matrix might be subsequently added to **S**. The default is extra_space_s = 0.

s_also_by_column is a scalar variable of type default LOGICAL, that must be set .TRUE. if the array S%ptr should be allocated and set to indicate the first entry in each column of **S**, and .FALSE. otherwise. The default is s_also_by_column = .FALSE..

space_critical is a scalar variable of type default LOGICAL, that must be set .TRUE. if space is critical when allocating arrays and .FALSE. otherwise. The package may run faster if space_critical is .FALSE. but at the possible expense of a larger storage requirement. The default is space_critical = .FALSE..

deallocate_error_fatal is a scalar variable of type default LOGICAL, that must be set .TRUE. if the user wishes to terminate execution if a deallocation fails, and .FALSE. if an attempt to continue will be made. The default is deallocate_error_fatal = .FALSE..

prefix is a scalar variable of type default CHARACTER and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string prefix(2:LEN(TRIM(prefix))-1), thus ignoring the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default prefix = "".

2.3.3 The derived data type for holding informational parameters

The derived data type BSC_inform_type is used to hold parameters that give information about the progress and needs of the algorithm. The components of BSC_inform_type are:

status is a scalar variable of type INTEGER(ip_), that gives the exit status of the algorithm. See §2.5 for details.

alloc_status is a scalar variable of type INTEGER(ip_), that gives the status of the last attempted array allocation or deallocation. This will be 0 if status = 0.

bad_alloc is a scalar variable of type default CHARACTER and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if status = 0.

max_col_a is a scalar variable of type INTEGER(ip_), that gives the maximum number of entries in a column of **A**.

exceeds_max_col is a scalar variable of type INTEGER(ip_), that gives the number of columns of **A** that have more entries than the limit specified by control%max_col.

time is a scalar variable of type REAL(rp_), that gives the total CPU time (in seconds) spent in the package.

clock_time is a scalar variable of type REAL(rp_), that gives the total elapsed system clock time (in seconds) spent in the package.

2.3.4 The derived data type for holding problem data

The derived data type BSC_data_type is used to hold all the data for the problem and the workspace arrays used to construct the Schur complement between calls of BSC procedures. This data should be preserved, untouched, from the initial call to BSC_initialize to the final call to BSC_terminate.

2.4 Argument lists and calling sequences

There are three procedures for user calls (see §2.6 for further features):

1. The subroutine BSC_initialize is used to set default values, and initialize private data, before solving one or more problems with the same sparsity and bound structure.
2. The subroutine BSC_form is called to form the Schur complement.
3. The subroutine BSC_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by BSC_form at the end of the solution process.

We use square brackets [ ] to indicate OPTIONALarguments.

2.4.1 The initialization subroutine

Default values are provided as follows:

```
CALL BSC_initialize( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type BSC_data_type (see §2.3.4). It is used to hold data about the problem being solved.

control is a scalar INTENT(OUT) argument of type BSC_control_type (see §2.3.2). On exit, control contains default values for the components as described in §2.3.2. These values should only be changed after calling BSC_initialize.

inform is a scalar INTENT(OUT) argument of type BSC_inform_type (see Section 2.3.3). A successful call to BSC_initialize is indicated when the component status has the value 0. For other return values of status, see Section 2.5.

2.4.2 The subroutine for forming the Schur complement

The sparse matrix $\mathbf{S} = \mathbf{A}\mathbf{D}\mathbf{A}^T$ is formed as follows:

```
CALL BSC_form( m, n, A, S, data, control, inform[, D] )
```

m is a scalar INTENT(IN) argument of type INTEGER(ip_) that specifies the number of rows of **A**. **Restriction:** m ≥ 0.

n is a scalar INTENT(IN) argument of type INTEGER(ip_) that specifies the number of columns of **A**. **Restriction:** n > 0.

A is a scalar INTENT(IN) argument of type SMT_type whose components must be set to specify the data defining the matrix **A** (see §2.3.1).

S is a scalar INTENT(OUT) argument of type SMT_type whose components will be set to specify the *lower triangle* of the Schur complement $\mathbf{S} = \mathbf{A}\mathbf{D}\mathbf{A}^T$. In particular, the nonzeros of the lower triangle of **S** will be returned in co-ordinate form (see §2.1.2). Specifically S%type contains the string COORDINATE, S%ne gives the number of nonzeros, and the array entries S%row(i), S%col(i) and S%val(i), i = 1, ..., S%ne give row and column indices and values of the entries in the lower triangle of **S** (see §2.3.1). In addition, for compatibility with other GALAHAD packages, S%m and S%n provide the row and column dimensions, m and n, of **S**.

data is a scalar INTENT(INOUT) argument of type BSC_data_type (see §2.3.4). It is used to hold data about the problem being solved. It must not have been altered **by the user** since the last call to BSC_initialize.

control is a scalar INTENT(IN) argument of type BSC_control_type (see §2.3.2). Default values may be assigned by calling BSC_initialize prior to the first call to BSC_solve.

inform is a scalar INTENT(OUT) argument of type BSC_inform_type (see §2.3.3). A successful call to BSC_solve is indicated when the component status has the value 0. For other return values of status, see §2.5.

D is a rank-one OPTIONAL INTENT(IN) argument of type REAL(rp_) and length at least n, whose i-th component give the value of the i-th diagonal entry of the matrix **D**. If D is absent, **D** will be assumed to be the identity matrix.

2.4.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL BSC_terminate( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type BSC_data_type exactly as for BSC_solve, which must not have been altered **by the user** since the last call to BSC_initialize. On exit, array components will have been deallocated.

control is a scalar INTENT(IN) argument of type BSC_control_type exactly as for BSC_solve.

inform is a scalar INTENT(OUT) argument of type BSC_inform_type exactly as for BSC_solve. Only the component status will be set on exit, and a successful call to BSC_terminate is indicated when this component status has the value 0. For other return values of status, see §2.5.

2.5 Warning and error messages

A negative value of inform%status on exit from BSC_form, BSC_solve or BSC_terminate indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

-1. An allocation error occurred. A message indicating the offending array is written on unit control%error, and the returned allocation status and a string containing the name of the offending array are held in inform%alloc_status and inform%bad_alloc respectively.

-2. A deallocation error occurred. A message indicating the offending array is written on unit control%error and the returned allocation status and a string containing the name of the offending array are held in inform%alloc_status and inform%bad_alloc respectively.

-3. One of the restrictions $n > 0$ or $m \geq 0$ or requirements that prob%A_type contain the string 'DENSE', 'COORDINATE' or 'SPARSE_BY_ROWS' has been violated.

-46 A row of **A** has more than control%max_col entries.

2.6 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable control of type BSC_control_type (see §2.3.2), by reading an appropriate data specification file using the subroutine BSC_read_specfile. This facility is useful as it allows a user to change BSC control parameters without editing and recompiling programs that call BSC.

A specification file, or specfile, is a data file containing a number of "specification commands". Each command occurs on a separate line, and comprises a "keyword", which is a string (in a close-to-natural language) used to identify a control parameter, and an (optional) "value", which defines the value to be assigned to the given control parameter. All keywords and values are case insensitive, keywords may be preceded by one or more blanks but values must not contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specfile is limited to 80 characters, including the blanks separating keyword and value.

The portion of the specification file used by BSC_read_specfile must start with a "BEGIN BSC" command and end with an "END" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by BSC_read_specfile .. )
  BEGIN BSC
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by BSC_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "BEGIN BSC" and "END" delimiter command lines may contain additional (trailing) strings so long as such strings are separated by one or more blanks, so that lines such as

```
BEGIN BSC SPECIFICATION
```

and

```
END BSC SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN BSC" and "END" delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is ! or * are ignored. The content of a line after a ! or * character is also ignored (as is the ! or * character itself). This provides an easy manner to "comment out" some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of three different types, namely integer, logical or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are "ON", "TRUE", ".TRUE.", "T", "YES", "Y", or "OFF", "NO", "N", "FALSE", ".FALSE." and "F". Empty values are also allowed for logical control parameters, and are interpreted as "TRUE".

The specification file must be open for input when BSC_read_specfile is called, and the associated device number passed to the routine in device (see below). Note that the corresponding file is REWINDed, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by BSC_read_specfile.

2.6.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL BSC_read_specfile( control, device )
```

`control` is a scalar `INTENT(INOUT)` argument of type `BSC_control_type` (see §2.3.2). Default values should have already been set, perhaps by calling `BSC_initialize`. On exit, individual components of `control` may have been changed according to the commands found in the specfile. Specfile commands and the component (see §2.3.2) of `control` that each affects are given in Table 2.1.

command	component of `control`	value type
`error-printout-device`	`%error`	integer
`printout-device`	`%out`	integer
`print-level`	`%print_level`	integer
`has-a-changed`	`%new_a`	integer
`maximum-column-nonzeros-in-schur-complement`	`%max_col`	integer
`extra-space-in-s`	`%extra_space_s`	integer
`space-critical`	`%space_critical`	logical
`deallocate-error-fatal`	`%deallocate_error_fatal`	logical
`output-line-prefix`	`%prefix`	character

Table 2.1: Specfile commands and associated components of `control`.

`device` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set to the unit number on which the specfile has been opened. If `device` is not open, `control` will not be altered and execution will continue, but an error message will be printed on unit `control%error`.

2.7 Information printed

If `control%print_level` is positive, information about the progress of the algorithm will be printed on unit `control%out`. If `control%print_level` ≥ 1, statistics concerning the formation of **S** as well as warning and error messages will be reported.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: `BSC_solve` calls the GALAHAD packages `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SMT`, `GALAHAD_QPT` and `GALAHAD_SPECFILE`,

Input/output: Output is under control of the arguments `control%error`, `control%out` and `control%print_level`.

Restrictions: `n` > 0, `m` ≥ 0, `A_type` $\in$ { `'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'` }.

Portability: ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

4 EXAMPLE OF USE

Suppose we form the Schur complement $\mathbf{S} = \mathbf{A}\mathbf{D}\mathbf{A}^T$ with matrix data

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & & \\ & & 1 & 1 \\ 1 & & & 1 \end{pmatrix} \text{ and } \mathbf{D} = \begin{pmatrix} 1 & & & \\ & 2 & & \\ & & 3 & \\ & & & 4 \end{pmatrix}$$

Then storing the matrices in sparse co-ordinate format, we may use the following code:

```
! THIS VERSION: GALAHAD 2.6 - 21/10/2013 AT 13:00 GMT.
   PROGRAM GALAHAD_BSC_EXAMPLE
   USE GALAHAD_BSC_double         ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
   TYPE ( BSC_data_type ) :: data
   TYPE ( BSC_control_type ) :: control
   TYPE ( BSC_inform_type ) :: inform
   INTEGER, PARAMETER :: m = 3, n = 4, a_ne = 6
   TYPE ( SMT_type ) :: A, S
   REAL ( KIND = wp ), DIMENSION( n ) :: D
   INTEGER :: i
   D( 1 : n ) = (/ 1.0_wp, 2.0_wp, 3.0_wp, 4.0_wp /)
!  sparse co-ordinate storage format
   CALL SMT_put( A%type, 'COORDINATE', i )     ! storage for A
   ALLOCATE( A%val( a_ne ), A%row( a_ne ), A%col( a_ne ) )
   A%ne = a_ne
   A%val = (/ 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
   A%row = (/ 1, 1, 2, 2, 3, 3 /)
   A%col = (/ 1, 2, 3, 4, 1, 4 /)
! problem data complete
   CALL BSC_initialize( data, control, inform ) ! Initialize control parameters
   CALL BSC_form( m, n, A, S, data, control, inform, D = D ) ! Form S
   IF ( inform%status == 0 ) THEN               !  Successful return
     WRITE( 6, "( ' S:', /, ( ' row ', I2, ', column ', I2,                  &
    &  ', value = ', F4.1 ) )" )                                             &
        ( S%row( i ), S%col( i ), S%val( i ),  i = 1, S%ne )
   ELSE                                         !  Error returns
     WRITE( 6, "( ' BSC_solve exit status = ', I6 ) " ) inform%status
   END IF
   CALL BSC_terminate( data, control, inform )  !  delete internal workspace
   END PROGRAM GALAHAD_BSC_EXAMPLE
```

This produces the following output:

```
 S:
 row  1, column  1, value =  3.0
 row  3, column  1, value =  1.0
 row  2, column  2, value =  7.0
 row  3, column  2, value =  4.0
 row  3, column  3, value =  5.0
```

The same problem may be solved holding the data in a sparse row-wise storage format by replacing the lines

```
!  sparse co-ordinate storage format
...
! problem data complete
```

by

```
! sparse row-wise storage format
   CALL SMT_put( A%type, 'SPARSE_BY_ROWS', i ) ! storage for A
   ALLOCATE( A%val( a_ne ), A%col( a_ne ), A%ptr( m + 1 ) )
   A%val = (/ 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! matrix A
   A%col = (/ 1, 2, 3, 4, 1, 4 /)
   A%ptr = (/ 1, 3, 5, 7 /)                 ! Set row pointers
! problem data complete
```

or using a dense storage format with the replacement lines

```
! dense storage format
   CALL SMT_put( A%type, 'DENSE', i )  ! storage for A
   ALLOCATE( A%val( n * m ) )
   A%val = (/ 1.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, 0.0_wp, 0.0_wp,          &
              1.0_wp, 1.0_wp, 1.0_wp, 0.0_wp, 0.0_wp, 1.0_wp /) ! A
! problem data complete
```

respectively.